Trends and Standards in Document Representation

Vania Joloboff

Bull Research Center. BP 68 - 38402 Saint Martin d'Hères Cedex. France

ABSTRACT

This paper starts by tracing the architecture of document preparation systems. Two basic types of document representations appear : at the page level or at logical level. The paper then focuses on logical level representations and tries to survey three existing formalisms : SGML, Interscript and ODA.

1. Introduction

Document preparation systems might be now the most commonly used computer systems, ranging from stand-alone text processing individual machines to highly sophisticated systems running on mainframe computers. All of those systems internally use a more or less formal system for representing documents. Document representation formalisms are very different according to their goals. Some of them define the interface with the printing device, they are oriented towards a precise geometric description of the contents of each page in a document. Others are used internally in systems as a memory representation. Yet others have to be learned by users ; they are symbolic languages used to control document processing.

The trouble is that there are today nearly as many representation formalisms as document preparation systems. This makes it nearly impossible, first to interchange documents among heterogeneous systems, second to have standard programming interfaces for developping systems. Standardization organizations and large companies are now trying to establish standards in the field in order to stop proliferation of formalisms and facilitate document interchange.

This paper focuses in the last sections on three document representation formalisms often called 'revisable formats', namely SGML [SGML], ODA [ODA], and Interscript [Ayers & al.], [Joloboff & al.]. In order to better understand what is a revisable format, the paper starts with a look at the evolution of the architecture of document preparation systems.

2. Architecture of document preparation systems.

Document preparation systems have appeared as soon as computer printing devices were able to output typewriter-like quality documents. Although the evolution of printing technology have been the major one, several factors have influenced the architecture of document preparation systems: low cost computing power, distributed systems, and the simple maturation of ideas in the field. The evolution of printing technology has lead to the digital representation of documents ready to be printed, called final form representation. The evolution of software techniques has principally lead to representations capturing the logical structure, the structure that is perceived by the author when the document is revised, i.e. constructed or modified.

2.1. Final form representation

On early document preparation systems, printing devices were basically typewriter-like terminals directly connected in character mode to the unique processing computer. Those devices were driven by sequences of control characters inserted in the data stream they received in order to produce layout rendition (underlining, overstriking). A formatting system basically had to translate the formatting commands into printer control sequences.

As printers from different vendors had different control sequences, transparency for the formatting system was needed to be able to print the same document on different sites with different printers. It was achieved by creating device independent formats. *Final form representation* had appeared, that is, the final digital representation of a document before it is printed. The main property of a final representation is that the number of pages in a document has then been computed. The way each object (character string or graphics) should appear on the page is totally determined.

As it becomes possible to introduce a processor into the printing device, yet keeping prices reasonably low (cheap chip), final form representations tend to become more and more complex. A final form representation is not any more a sequence of characters but an organized structure. A formal method is used to describe the page layout. For non impact printers, formalisms are needed that allow for the representation of images and graphics.

Virtually any image is reproducable on non impact printers: characters in any alphabet, graphics and images as well. There do not exist any more a specific set of imaging functions available from the hardware. Then the limit to the expressive power of the page creator is set by the *software interface*.

This fundamental change brought by technology has implied a fundamental change in the design of final form representation. Non-impact printing has introduced a new generation of final form representation formalisms, highly structured, trying to offer a maximum expressiveness. They aim at universality,

that is, they allow for the description (in theory) of any page for any printer. They divide into static formats and dynamic ones, more recent. In a static format the page layout is described as a static data structure. The standard CCITT T73 [T73] is a typical example of such formats. In dynamic formats, also referred to as procedural page description languages, a page description actually describes *how to compute* the layout.

Brian Reid's paper [Reid86] in that very conference talks more extensively on procedural page description languages, such as PostScript[PostScript]. The point we want to emphasize now is that the architecture (figure 1) of document preparation systems has now a clean interface with printing devices. It generates a final form representation of documents in terms of a structured page description formalism.

2.2. Revisable form representation

A document has to undergo many additions or modifications before it is ready to be printed. Working on a page based representation when editing a document would be tedious and cumbersome both for users and the editing system. An unformatted representation of documents is necessary. This representation typically is the output of the editing system and the input of the formatting system. Figure 1 shows the three basic components of a document preparation system: editing, formatting and printing. Revisable form and final form are the two representations interfacing these components.

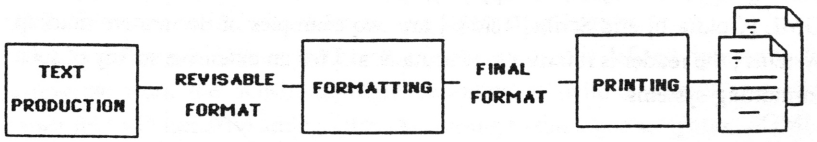


Figure 1. Typical architecture of a document preparation system.

The first document preparation systems have naturally imitated the method used in the publishing industry for typesetting: additional information is interspersed among the document contents to produce a data stream directly processed by the typesetting device. On those early systems the revisable form representation simply consists of a text file containing control sequences, directly keyed in by the user from a standard terminal.

Control sequences consists of a series of *markup signs*. That was the beginning of so called *procedural markup languages*, since those markup signs were interpreted as instructions controlling subsequent processing in the formatting system.

Procedural markup has well known inconvenients:

- the logical structure of a document is not much evidenced once the document is marked up. For example, if chapter titles have been marked with a centering command, it does not appear clearly that what follows a centering command is a title. If someone later wants to flush all titles right, changing all centering commands into flush commands will probably not give the expected result.

- the style of the resulting documents, i.e. the aspect of the document layout, is determined by the user who placed the markup signs. A good layout style, if some style at all, requires from the user some typographic knowledge. The lack of this knowlege is responsible for all of the ugly documents produced on procedural markup systems... Also, it makes it difficult to output the same document in a different style.

Disavantages of procedural markup have been avoided with a new method, known as *declarative markup*. The standpoint in declarative markup is that the user should describe the logical structure of a document, what is to be processed rather than how the document content is to be processed. A user enters mark up signs indicating logical properties of data, for example *paragraph* or *heading*, expressing its logical structure, which sounds more familiar, and does not imply a particular processing. The responsibility of making consistent styles, or applying specific functions is left to the system. GML [Goldfarb] and Scribe[Reid83] are two examples of declarative markup systems ; the reader is referred to [Furuta & al.] for an extensive survey of such formatting systems.

The SGML formalism is essentially the definition of an international standard by ISO for covering these systems. Yet a SGML entity may refer to non-character data, as shown in the next section, it has been designed in the spirit of all markup systems. As the standard says (page 3) "The millions of existing text entry devices must be supported. SGML documents can easily be keyboarded and understood by humans."

A user does not need a specific editor to build a markep up document. As far as there are only characters, any editor will do on any standard terminal. The revisable form representation of a document in a markup system, be it declarative or procedural, is (or should be) fully known from users, they have to key it in...

More recent approaches have a different viewpoint. They assume the revisable form representation is not directly accessed by users, but solely by the editing system. Thus a specific editor is needed, which generates that

representation. It is intended such editors will not expose users to the revisable representation; that they will actually hide to the user the internal representation of documents, constructing themselves this representation from the user input.

These editors are expected to provide a more convivial user interface. Most of the editors from this new generation do not run on standard terminals, for example Grif, presented in this conference [Quint & Vatton]. They rather use bitmap display terminals, a window system and a pointing device.

The new type of document representation used in this approach may then be designed to be quite complex, nearly unmanageable by human beings, but very suitable to be handled by computers. Graphics and images may be directly inserted in documents more easily than for markup formats. Graphics may rely on existing standard graphics representation, images may be stored trough specific data compression techniques, while the user only sees on the screen a real layout.

Interscript and ODA both belong to this new genereration of formalisms. They assume more computing power from the editing system, they lose the possibility to be directly entered from a standard terminal, but promise many more possibilities.

3. Generalized Markup Language

SGML stands for Standard Generalized Markup Language. It is essentially a declarative markup language, which has inherited mainly from its ancestor GML. However it includes a lot of new interesting features.

A first difference with its predecessors is that markup is defined rigorously. It is possible from the SGML standard definition to build a general syntactic parser that will not arise ambiguities. According to this rigorous syntax, SGML documents may be processed very much like programs by a compiler. A document may be parsed to build an abstract syntactic tree together with its attributes.

Semantics of that tree may be evaluated by semantic functions according to the attributes values. Thus, SGML can be used for other tasks than formatting ones. Semantics of markup tags and attributes might be used for machine translation, automatic indexing or any other process needing parsing of documents.

A markup sign in SGML is named a tag. Any element which needs to be tagged starts with a start-tag and ends with an end-tag. Any tag is delimited by the characters '<' and '>'. A tag is defined by an identifier, which appears first in the start-tag. An end-tag repeats the same identifier preceded by '/'. Note that all of these mark characters are redefinable for each document. End tags may also be omitted under conditions specified in the standard. For

example, a paragraph will appear as :

 This is a short paragraph.

A drawback of usual declarative markup systems is that one is forced to use the catalog of markup tags which is offered by the system. Since markup tags express the logical structure of documents, it means one cannot define the logical structure in other terms than the general tags set up once for all by the system.

A property of SGML is that tags are themselves described trough a formal language: the SGML meta-language, which may be used within SGML documents to dynamically define new symbols. Syntax to introduce a meta-language construct simply follows '<' by '!'.

The SGML meta language allows for the definition of complex constructs, named *elements*. An element declaration defines of a class of objects, i.e. an element type. Subsequent objects in the document may be tagged with the element name. Elements may have a hierarchical structure, and each element in the hierarchy may have its own attributes. Element types may be used either to facilitate the interactive creation of documents, to control the validity of a document structure, or to associate a layout style to a particular document type.

For example, one might define a document type for a conference paper as follows:

```
<!ELEMENT
 1  paper                (title abstract sections)
                language   CHARS
 2  title                (#CDATA)
 3  abstract             (p)
 4  body                 (p*)
 >
```

This document type declaration specifies that a paper has a title, an abstract, and a body. The title consists of characters, the abstract is one paragraph and the body one or more paragraphs. A paper has a language attribute to indicate in which language it is written. More complex combinations can be designed to define document types that have some commonality.

The facility to define new elements brings troubles when laying out those elements, because the formating system then does not know how to format such constructs. SGML provide two ways for handling that situation. The first one is naturally to add to the SGML system a procedure to take care of the new tags. This requires a good knowledge of the system and prohibits further interchange of documents with such tags to systems which do not have this

procedure. The second one is to use a LINK tag. A LINK tag says to the system that a construct should be handled as another one, presumably known from the system, with possible attributes modifications. For example, if one says <!LINK abstract paragaph indent=5>, it means an abstract has to be formatted like a paragraph, however using a different indentation value.

It is often required in a document to be able to refer to other parts of the document. Some binding mechanism is needed in the formalism to attach a value to some identifier, which resembles to progamming language variables. Binding is achieved in SGML trhough *entity declaration* and *entity references*. An entity (a value, a character string or any valid SGML constituent) may be bound to a name by the notation <!ENTITY name entity value>. From now on, that entity may later be referenced by its name either to set an attribute value, or to be included into the running text. Entities also provide means to handle non character data. An external entity is declared <!ENTITY name SYSTEM system information>. Then it is known that this entity is not in the document stream. The processing system will find in the system information how to access that content.

If the document is to be interchanged among different computers with different operating systems, this system information is specific to each system. SGML provides an IGNORE/INCLUDE mechanism for that purpose. Information relative to some particular system, let say osx, has to be encoded within the magic declaration <![osx;[<?commands for osx system>]]>. Then a user only needs to turn a switch at the beginning of the document to the local system for the document to be processed correctly.

4. Interscript

We mentioned previously Interscript is a representation formalism from a new generation. Interscript, which was originally designed at Xerox PARC, starts from the idea that a document representation should be suited to be processed by computers, not by the humans who manipulate documents.

Such things as traversing trees, evaluating expressions, searching values of variables within contexts are among what computers can easily do. Thus, a fundamental notion in Interscript is to rely on a formal language to describe document constructs, not only a document logical structure, but all formal constructs that could be necessary into a document representation. These abstract constructs may be data structures such as paragraphs, fonts, geometric shapes, but may also represent computations, like setting a context or evaluating expressions within some context.

The Intescript approach is very much like the approach used in software engineering: general programming languages are used by people to build abstract constructs and procedures to solve their particular problem. A

document representation problem should be solved using the a document representation language. The Interscript base language is simple (around 25 grammar rules) and powerful. Its semantics are well defined but its syntax rapidly leads to document that cannot be managed by humans.

A document encoded in the Interscript base language is called a script. A script is very much like a program. The processing paradigm (figure 2) is that a script should be first internalized by a system. Internalizing a script implies *execution of computations*, which are dictated only by Intercript base language semantics, and result in the construction of another representation available for the client process.

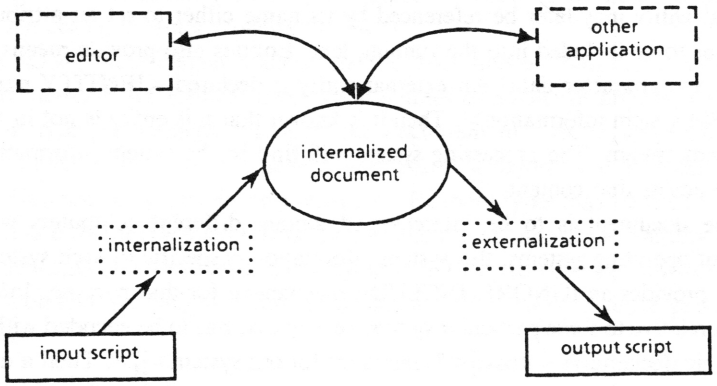


Figure 2. Interscript processing model.

This simply means that one translates a standard disk representation into a non standard memory representation, while achieving computations.

Computations are necessary in the internalizing process because the base language includes a binding mechanism and the evaluation of expressions within hierarchical contexts. For example, evaluating the expression :

rightmargin = leftmargin + linelength

needs to obtain the values bound to the variable names.

We will not in this paper enters into the details of the internalizing process, which looks like the evaluation of any interpreted programming language, to focus on the central concepts of *node* and *tag*.

A script is a hierarchy of nodes. Nodes have contents and tags. The authors have compared an Interscript node to a bottle of wine. The contents of the bottle is qualified by several tags on the bottle: a price tag, a product number tag. Interscript tags similarly qualifies the node contents. To some extent an Interscript tag is similar to an sgml tag, it introduces an element, it has

attributes, it denotes structural properties of the contents.

The difference is that, first a Interscript node may have simultaneous tags, second attributes of a tag may be bound to an expression which must be evaluated. For example, a figure caption could be affixed with both a CAPTION and a PARAGRAPH tag. The paragraph tag says that the caption text has to be laid out as a parapragph, the caption tag restricts the placement of that paragraph relatively to the figure picture. The leftmargin attribute of the paragraph might be set to be equal to the margin of some object X. Then the node hierarchy is searched for that X.

Interscript syntax denotes nodes between curly braces. Tags are character strings followed by a dollar sign. A typical node is :

```
{ PARAGRAPH$ PARAGRAPH.leftmargin = 10
    {CHARS$ <paragraph text content> }}
```

4.1. The pouring process

Markup languages do not provide good support for describing layout. They start from idea that a user should hardly be able to specify layout, in order to enforce style discipline. It is true that users of a document preparation system are usually not interested in setting line and page breaks, selecting fonts, positioning titles, etc. However, they are often concerned with placement of logos, page numbers, whether there are one or more columns ; what we might call macroscopic layout.

Interscript provides for that purpose a comprehensive mechanism we shall name *descriptive layout*. Descriptive layout does not prohibits the use of styles, it would rather enforce their use too, however it allows for the specification of high level layout. Tags have been defined which symbolically represent the layout process. By placing those tags at appropriate places and specifying attributes values, a user may indicate to a formatting process how layout should be achieved.

All of those specifications appear as parameters of the Interscript *pouring process*. The Interscript metaphor for this process is that the document content, is *poured* into some *liquid layout*, resulting in a *solid layout*. Liquid layout basically serves as a template which guides the pouring process in its actions. The pouring process is naturally described by means of constructs expressed in the base language. The fundamental pattern for invoking a pouring process is:

```
{ POUR$
    POUR.template = {TEMPLATE$ -- template -- }
    contents to be poured }
```

A template basically is a hierarchy of boxes. A box defines a rectangular area on which constraints apply to locate it relatively to other boxes. Assume a user

wants a page layout as shown on figure 3. That page has a header at the top and a logo down the header. The content should be laid out in a right area on the page, leaving some place for margin comments that should be placed on the left.

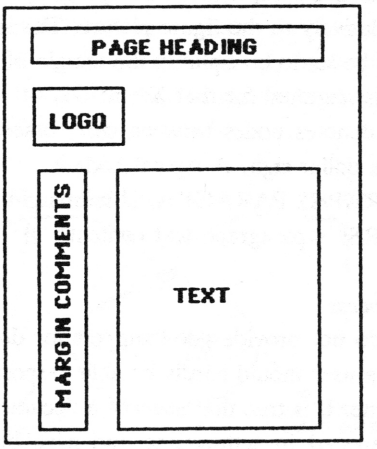


Figure 3. A page layout.

The template for that page will be a hierarchy whose root node is a page node. This page contains four boxes respectively specifying the position of the heading, logo, margin comments and text. The position constraints are those indicated by the user.

When the content is poured in the page, the pouring process must not pour any content into the heading or the logo box, neither does it pour text content into the margin comment box. This correct placement of data is ensured by the MOLD mechanism.

When a box is to receive data, it shows a MOLD tag accompanied by a label. The pouring process does not try to pour content into boxes that do not have a mold tag, it directly places them within the page box. If a box is a mold, then it looks in the node contents for some node with a matching label. All content portions with matching label are to be poured into that box.

It may be the case that there is too much content to be poured to fit on a single page. A template may specify that it is a sequential template, an iterative or an alternative one. In a sequential template, the pouring process will consider all boxes in the hierarchy sequentially. If a template is full, or no more matching content exists, it considers the next mold. An iterative template will repeat itself until all matching content has been poured.

The argument in favour of that unique representation seems to be that most editing systems have to manage both structures. The standard says (part 2 - page 75)

> "In a text processing system with separate editing and formatting subsystems, the specific layout is created after any changes to the specific logical structure and content have been made. In a word processor type editor, small editing changes may be incorporated directly into existing specific layout structure after every command, without recreating the entire specific layout structure."

This issue is discussed in the conclusion. This section only tries to present the ODA formalism. Figure 5 shows the main constituents of a document. It has six parts, a document profile, a document style, a generic and a specific logical structure, a generic and a specific layout structure.

An alternative template specifies different possibilities for pouring the content, the layout process is responsible for choosing one. One possibility is to try all of them, and pick up the best by its own criteria. Another possibility is to have additive tags indicating when or how to select an alternative. For example, one might indicate a template to be used on screens, another one for paper.

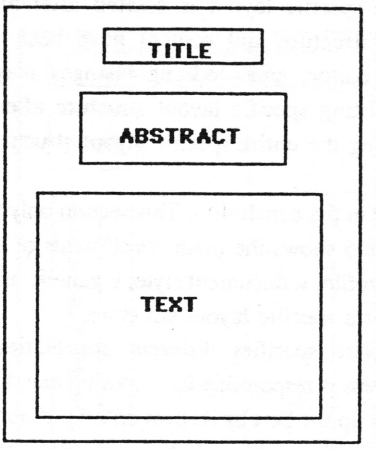
Templates may also be combined. Figure 4 shows an typical example, a paper like this one. The first page has a particular layout showing the title, authors, an abstract of the paper ; down of the abstract starts the paper. All subsequent pages are in the same format, showing only a page heading and text. The template for that document is a sequential one. It contains the first page as a single box (a page is a particular box), next an iterative page template.

Many others possibilities are offered by this descriptive layout process. We only described in this paper its general properties.

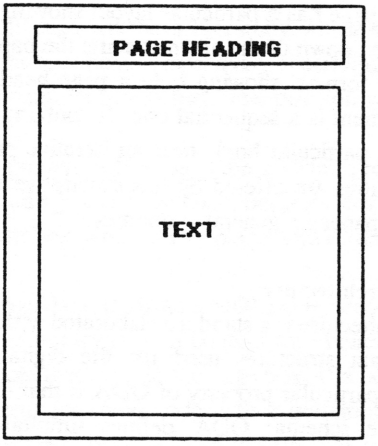
5. Office Document Architecture

Office Document Architecture is a standard elaborated within ISO to introduce a standard in the data structures used for the digital representation of documents. The most particular property of ODA is that it does not fit in the traditional architecture schema: ODA defines simultaneously the logical structure and the layout structure of a document, i.e. it is both a logical and a page description format.

A document may actually contain only one of the structure. This is indicated when transmitting in the document profile. The document profile also contain data related to the whole document: creation date, last alteration date, originators, status, etc.



FIRST PAGE



OTHER PAGE

Figure 4. Two different page layout in a single template.

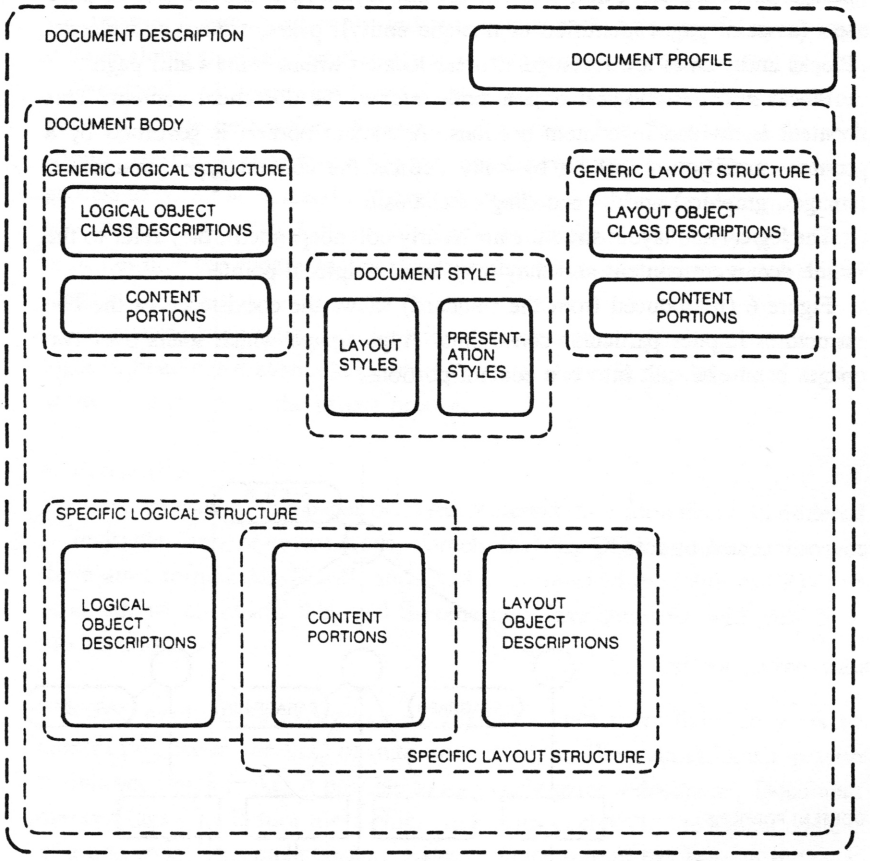


Figure 5. ODA document components.

Generic and specific are to be interpreted respectively as class and instance. For example a generic structure named *conference paper* will describe the general structure and properties of a conference paper, as shown in the SGML section. A particular instance of a conference paper will be described by its specific structure. Attributes defined in the generic will be valued in the specific structure, possibly to a default value specified in the generic part. The specific structure should be consistent with the generic one, and will probaly inherit properties from the generic structure. Specific logical and layout

structure are trees whose leaf nodes are named basic objects and other nodes composite objects. Any node may carry attributes.

The specific logical structure expresses the structure of the document in, e.g. paragraphs, chapters, titles, etc. The specific layout structure is a tree of page sets (a set of pages identified as a single entity), pages, frames and blocks. Blocks and frames are rectangular areas located within frames and pages.

Blocks and basic logical objects both refer to the document content. This content is divided in content portions. A content portion is governed by a content architecture, which basically defines the content type (characters, images, graphics) and its encoding mechanism.

The logical and layout structure are clearly not independent, they refer to the same document content and they may have reciprocal pointers.

Figure 6 (reproduced from the standard) shows the coexistence of the two structures implies particular constraints. A paragraph which spans over two pages has to be split into two content portions.

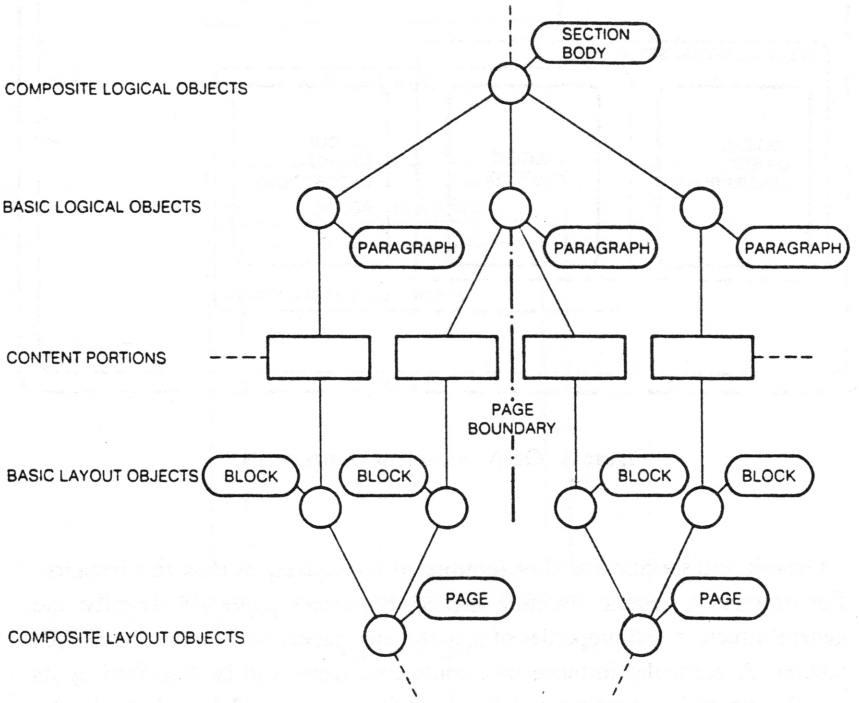


Figure 6. Simultaneous layout and logical structure.

Styles simply are a named set of attributes, which can be referenced from other components in the document. They divide in layout style and presentation style. A presentation style is attached to a basic object and depends upon the nature of that objects. For characters, it would indicate font information, for images it would probably indicate colors or half-toning. Layout style defines global style information. It can be referenced only from logical objects.

The standard is somewhat fuzzy about generic structures. There is no clause devoted to the description of generic structures, while there is one for each specific structure. It says that object class description are used by the editing process to construct a specific logical structure but it does not say much about such descriptions. Part 3 of the standard, which describes the layout process, indicates how the generic layout structure should be used, hence give a clearer idea.

The generic layout structure contains common content portions, for example logos or headings that should be used in many places in the document. It also serves as a guide for the layout process.

6. Conclusion

We have focused in this paper on three representation formalisms considered as revisable form representations, namely SGML, ODA and Interscript. On these three formalisms, SGML and ODA have reached the status of ISO draft proposal, which means they will become definitive standards with very little modifications.

SGML results from experience accumulated since more than ten years by current practice in the field of markup languages. The standard has a precise definition, which makes it possible to rigorously parse a document. Document markup tags may induce hierarchical structures expressing the logical content of a document. A simple binding system among entities has been introduced, which allows for cross referencing among entities.

Knowing that SGML has been running on many machines, that high quality text books have been produced through an SGML system, A vendor commercializing a markup document system would probably better take SGML rather than inventing a new formalism.

ODA did not follow the same standardization process as SGML. ODA is an attempt by an ISO working group, consisting mostly of text processing system vendors representatives, to define a standard before there are hundred of representation formalisms around the world that would not be compatible. Hence there is no current practice of ODA and it is only expected that most of new systems will use ODA. However there are a few objections to actually

using ODA.

The choice of the two coexisting layout and logical structures. might lead to implementation problems. The content portions which have to be split to satisfy layout constraints will have to be recollected when the layout is modified.

Now that most printing device vendors have upgraded their machines to have a procedural page description language, the design of the layout structure in terms of frames and blocks looks old fashioned and contradictory with the fact that ODA claims to be a standard for future systems.

One might fear too with ODA that vendors will actually offer "ODA subsets". Particularly it is the case SGML can be considered as such a subset. The standard explicitly states, probabaly for reasons of compatibility among ISO standards, that an SGML document may be transmitted within an ODA document (part5-page 4):

> "Any subdocument within a (ODA) document may be represented either by descriptors and text units or by an SGML entity set. An SGML entity set is a self contained unit of SGML information, which is denoted but the term document in the SGML standard."

A vendor may actually sell an SGML system as an ODA subset system. If each vendor offers a subset of ODA, it might be the case that all of those systems will be actually incompatible, which is not desirable for an interchange standard. This argument is naturally true for all standards, but ODA design makes it easier to have closed subsets.

For example, it is possible to design an ODA editing system that would not take into consideration all of the layout part and restrict to the logical structures and styles. This editing system will output only documents with logical structures and styles. These documents may be interchanged using the ODA format.

It is possible too, in terms of delimiting an ODA subset, to design a simple word processing machine that would use no logical structure at all to produce office documents with only a layout structure. But both of those ODA systems will not be able to interchange a single document.

Morover, it seems from ODA complexity that a complete ODA system can hardly be implemented on a small word processing workstation. Implementors of these relatively small workstations, who are willing to manage documents with both logical and layout structure, will probably have to define a subset in order to maintain satisfying performance.

Though SGML has been designed so that human beings may enter markup tags into a document, it might well be used as an internal representation for an editor that would not appear to the user as a markup system. Then the structuring possibilities offered by SGML may be used by the implementors to

represent complex internal structures, producing equivalent facilities to those of ODA. Documents produced by such an editor could hardly be revised by humans from a standard terminal, but they could still be output with the high quality of an SGML formatting system. Thus a vendor who is willing to implement a document preparation system has to choose among two international standards.

Interscript is not an international standard and it seems it will not become. The reason might be that Interscript design is too much a departure from their existing formalisms to be accepted by most vendors, who are mostly interested in standards. Remember that bitmap displays were developed at Xerox PARC in 1975. In 1985, still very few vendors offer text processing systems with a bitmap display and a pointing device. Interscript was also born at Xerox PARC in 1983 [Ayers & al], as a result of several years of experience with powerful text processing systems running on bitmap displays...

Yet Interscript will not be a standard in the eighties, it has introduced two important ideas, the notions of base language associated with an internalization process, and descriptive layout, which should be retained by people who are participating to the design of a new generation of document preparation systems.

Interscript proves that a base language can be defined which encompasses all abstractions that can be found in the document preparation world. It can describe as well a document logical structure, properties and structure of various kind of entities (font, paragraph, etc), and functional symbolisms like the pouring operation.

A base language considerably simplifies the software development of systems once it is implemented, but all over it gives cleanness to the systems and clarity in concepts. The Interscript base language is certainly not perfect. It can be improved and it might be actually too powerful for its goals.

Similarly, the idea of a layout process formally described and specified by abstract constructs, can be expressed in other terms than the particular Interscript pouring process. But both concepts have opened a direction for present research in the field.

References

[1] Adobe Systems Incorporated (1984). PostScript language manual. 1870 Embarcadero Road, Palo Alto California.

[2] Ayers, R.M., Horning, J.T., Lampson B.W, Mitchell J.G. (1984). Interscript : A Proposal for a Standard for the Interchange of Editable Documents. Xerox Palo Alto Research Center. 3333 Coyote Hill Road,

Palo Alto, California.

[3] CCITT (1984). Recommendation T73. Document interchange protocol for the telematics services.

[4] Furuta, R., Scotfiled, J., Shaw, A. (1982). Document Formatting Systems: Survey, Concepts, and Issues. ACM Computing Surveys, **14**, 3, pp 417-472.

[5] Goldfarb, C.,(1978). Document Composition Facility : Generalized Markup Language (GML) User's Guide. Technical report SH20-9160-0. IBM General Products Division. ACM Computing Surveys, **14**, 3, pp 417-472.

[6] International Standard Organization/ TC 97/ SC 18 (1985). Information Processing. Text and Office Systems - Document Structures. Draft Proposal 8613

[7] International Standard Organization/ TC 97/ SC 18 (1985). Information Processing. Text Preparation and Interchange. Processing and Markup Laguages. Draft Proposal 8879

[8] Joloboff, V., Pierce, R., Schleich, T. (1985). Document Interchange Standard "Interscript" International Standard Organization/ TC 97/ SC 18. Information Processing. Document N439R

[9] Reid, B.K. (1983). Scribe: Histoire et evaluation. Actes des journees sur la manipulation de documents, INRIA/IRISA, Rennes, France.

[10] Reid, B.K. (1986). Procedural Page Description Languages Proceedings of the conference on text processing and document manipultation, Notthingham, England.

[11] Quint, V., Vatton, I. (1986). GRIF: An interactive system for structured document manipulation. Proceedings of the conference on text processing and document manipultation, Notthingham, England.